Multiple Mutation Testing for Timed Finite State Machine with Timed Guards and Timeouts

Omer Nguena Timo, Dimitri Prestat, Antoine Rollet

Multiple Mutation Testing for Timed Finite State Machine With Timed Guards and Timeouts *

Omer Nguena Timo[1], Dimitri Prestat[2], and Antoine Rollet[3]

[1] CRIM - Computer Research Institute of Montréal
Montréal, Canada
omer.nguena-timo@crim.ca
[2] UQAM - University of Québec in Montréal
Montréal, Canada
prestat.dimitri@courrier.uqam.ca
[3] LaBRI, Bordeaux INP, University of Bordeaux
Bordeaux, France
antoine.rollet@labri.fr

Abstract. The problem of generating tests detecting all logical and timing faults which can occur in real-time systems is challenging; this is because the number of (timing) faults is potentially too big or infinite. As a result, it might be time consuming to generate an important number of adequate tests. The traditional model based testing approach considers a fault domain as the universe of all machines with a given number of states and input-output alphabet while mutation based approaches define a list of mutants to kill with a test suite. In this paper, we combine the two approaches by developing a mutation testing technique for real-time systems represented with deterministic timed finite state machines with timed guards and timeouts(TFSM-TG). In this approach, fault domains consisting of fault-seeded versions of the specification (mutants) are represented with non-deterministic TFSM-TG. The test generation avoids the one-by-one enumeration of the mutants and is based on constraint solving. We present the results of an empirical proof-of-concept implementation of the proposed approach.

1 Introduction

This paper deals with mutation testing of timed systems. Traditional model-based testing approaches consist of using the model of the specification as a base for the generation of a test suite, then applying it on the implementation of the system under test. This is a black-box testing, meaning that the implementation is considered as unknown, but with the assumption that it may be described by a formal model. It is the so-called test hypothesis. Since it is generally impossible to generate an exhaustive test suite, several approaches have been proposed to limit the size of test cases while providing enough confidence.

* This work was partially supported by MEI (Ministère de l'Économie et Innovation) of Gouvernement du Québec.

Concerning FSM model-based testing, one may consider coverage criteria of the specification: the test may be satisfying if for instance all the transitions or all the states are covered by the test suite. Another approach consists of obtaining a sufficient fault coverage: the test suite is guaranteed to detect all faults of a specific nature, e.g. output faults (the output provided by a transition is not correct) or transfer faults (the arrival state of a transition is not correct). As an example, the so called W-method [6] is known to detect both transfer and output faults whereas TT-method [16][4] focuses only on output faults. Generally, fault coverage based methods consider the traditional fault domain as the universe of all machines with a specific alphabet and a given number of states. On the other side, mutation testing is a technique originally developed to verify if a given test suite has a satisfying detecting power. In model-based testing, the principle consists of applying a certain number of small variations, called mutations, on the specification, then to check if these mutations are detected by the test suite. It is generally called "killing the mutants". Note that mutation testing techniques are also used in code based testing, but in this case, the mutations are applied directly in the source code. Dealing with real-time systems increases the difficulty of testing. For instance, faults related to timing errors should also be detected by a test method. For timed model based testing, many different models and conformance relations have already been proposed in the past. Some of them use a discrete representation of the time in the specification model, such as synchronous models [4,23,13], some other methods extend the so-called *ioco* theory [25] by defining a new conformance relation adapted to timed automata [1] with inputs and outputs [12,10,15,2,19,3]. Another category extends the FSM based testing theory using timed extensions of the FSM model and an adapted fault model [11,14,7,28].

The work in [9] proposes a method to generate a test suite with guarantee fault coverage for partial deterministic TFSM extended with timed guards only. TFSM with timed guards and timeouts (TFSM-TG) can express timed behaviors which cannot be expressed with TFSM with timeouts only [5]. More recently [26] proposes a method to generate tests detecting all the nonconforming implementations of an initialized TFSM with timeouts and TFSM-TG. The implementations belong to the traditional fault domain and they can have more states than the specification, up to a given number. The tests are generated by applying the W-method on an (possible huge) abstraction of TFSM-TG with classical FSM and transforming the resulting abstract tests into timed tests.

In this paper, we combine fault coverage based approaches and mutation testing applied to real-time systems. We propose a mutation testing technique for real-time systems represented with complete and deterministic TFSM-TG. We derive complete test suites from mutation machines. A mutation machine can be designed to represent the traditional fault model; but it can also represent customized fault models as well. Customized fault models could be parts of the traditional fault model and they can be covered with a reduced number of tests. The tests generated for the traditional fault model remain valid to test

[4] without special *status* input

customized fault models. However they could suffer from redundancy and could include useless tests. Running implementations with tests is time consuming. Then, reducing the number of tests to apply is really useful. This is the main motivation of our work. Our test generation approach is based on constraint resolution. It uses the distinguishing automaton of the specification and the mutation machine to determine revealing combs. They characterized the undetected mutants and serve to encode them with Boolean formula. We use a solver to check the satisfiability of the formula and conclude about the completeness of test and generate new tests if needed. We propose an abstraction of timed states of TFSM-TG and we use it to build the distinguishing automaton. This paper extends the approach proposed in [21,17] in a timed context. This is the main contribution of our paper. It is adapted to be applied on the TFSM-TG model. This contribution includes the abstraction of timed states, the definition of the distinguishing automaton for TFSM-TG, and the representation of the transitions passed during executions of TFSM-TG with the so-called combs.

The paper is organized as follows. The next section introduces the general theoretical background, a fault model for TFSMs-TG and the coverage of fault models with complete test suites. Section 3 characterizes detected mutants with revealing combs. In Section 4 we introduce an abstraction for executions of TFSM-TG and we define the distinguishing automaton. Section 5 presents an encoding of undetected mutants with Boolean formulas. We present a method for the test analysis and a method for the complete test suite generation in Section 6; we also present the results of an empirical proof-of-concept tool. We conclude the paper in Section 7.

2 Preliminaries

A timed guard π is an interval $\pi = \langle l_\pi, u_\pi \rangle$ where l_π is a non-negative integer, while u_π is either a non-negative integer or ∞, $l_\pi \leq u_\pi$, and the symbols $\langle$ and $\rangle$ represent $]$ or $[$. For example, the interval $[0, 5[$ contains all the non-negative real numbers smaller than 5. We let Π denote the set of timed guards. Given a timed guard $\pi = \langle l_\pi, u_\pi \rangle$ and a real number x, we define $\pi^{-x} = \langle max(0, l_\pi - x), max(0, u_\pi - x) \rangle$.

2.1 Timed FSM with timed guards and timeouts

Definition 1. *A* timed finite state machine with timeouts and timed guards [5] (TFSM-TG) *is a 6-tuple* $\mathcal{S} = (S, s_0, I, O, \lambda_S, \Delta_\mathcal{S})$ *where* S*,* I *and* O *are finite non-empty set of* states, inputs *and* outputs*, respectively,* s_0 *is the* initial *state,* $\lambda_S \subseteq S \times I \times \Pi \times O \times S$ *is an input/output transition relation and* $\Delta_\mathcal{S} \subseteq S \times \mathbb{N}_{\geq 1} \cup \{\infty\} \times S$ *is a timeout transition relation.*

Remark that we allow defining multiple timeout transitions in states of TFSM-TG, in the opposite of [5]. Later this will be used to compactly represent sets of implementations of a TFSM-TG specification. $\mathcal{S}$ in state s fires an input/output transition (s, i, π, o, s') to reach the transition's *target state* s' and produces output o if input i is applied in transition's *starting state* s when timed guard π is respected; π expresses the minimal and the maximal delays in s to fire the transition. It fires a timeout transition $(s, \delta, s') \in \Delta_{\mathcal{S}}$ defining timeout δ in s and reaches s' if no input is applied in s before δ expires. Let $\delta_{max}(s)$ denote the maximal (possibly infinite) timeout defined in state s. We require that the maximal finite constant in guards of the transitions defined in every state s should be smaller than $\delta_{max}(s)$. Because multiple timeouts can be defined in the same state, $\mathcal{S}$ necessarily fires a timeout transition defining $\delta_{max}(s)$ if no input is applied at s before $\delta_{max}(s)$ expires. A clock measuring the amount of the time elapsed in every state is implicitly reset when transitions are fired.

A *timed state* of TFSM-TG $\mathcal{S}$ is a pair $(s, x) \in S \times \mathbb{R}_{\geq 0}$ where $s \in S$ is a state of $\mathcal{S}$ and $x \in \mathbb{R}_{\geq 0}$ is the current value of the clock and $x < \delta$ for some $\delta \in \mathbb{N}_{\geq 1} \cup \{\infty\}$ such that $(s, \delta, s') \in \Delta_{\mathcal{S}}$. The initial timed state of $\mathcal{S}$ is $(s_0, 0)$.

An *execution step* of $\mathcal{S}$ in timed state (s, x) corresponds either to the time elapsing or the firing of an input/output or timeout transition; it is *permitted* by a transition t of $\mathcal{S}$. Formally, a tuple $((s, x), a, t, (s', x')) \in (S \times \mathbb{R}_{\geq 0}) \times (((I \times O) \cup \mathbb{R}_{\geq 0}) \times (\lambda_{\mathcal{S}} \cup \Delta_{\mathcal{S}})) \times (S \times \mathbb{R}_{\geq 0})$ is an execution step if it satisfies one of the following conditions:

- (timeout) $t = (s, \delta, s') \in \Delta_{\mathcal{S}}$, $a \in \mathbb{R}_{\geq 0}$, $x + a = \delta$ and $x' = 0$
- (time-elapsing) $t = (s, \delta, s'') \in \Delta_{\mathcal{S}}$, $a \in \mathbb{R}_{\geq 0}$, $x + a < \delta$, $x' = x + a$ and $s' = s$
- (input/output) $t = (s, i, \langle l, u \rangle, o, s') \in \lambda_{\mathcal{S}}$, $x \in \langle l, u \rangle$, $a = (i, o)$ with $(i, o) \in I \times O$ and $x' = 0$

In the time-elapsing step, the target state s'' of t and s' can be different. This is because the timeout δ is not expired and t is not fired; $\delta_{max}(s)$ is never exceeded.

An *execution* of $\mathcal{S}$ in timed state (s_0, x_0) is a finite sequence of steps $e = stp_1 stp_2 \dots stp_n$ with $stp_k = ((s_{k-1}, x_{k-1}), a_k, t_k, (s_k, x_k))$, $k \in [1, n]$ such that stp_1 is not an input/output step and stp_k is an input/output step implies that stp_{k-1} is a time-elapsing step for every $k \in [1..n]$. If needed, the elapsing of zero time unit can be inserted before input/output steps which are not immediately preceded with a time-elapsing step. The *timed input/output sequence* of execution e is the sequence $(i_1, o_1)d_1(i_2, o_2)d_2 \dots (i_l, o_l)d_l$ in $((I \times O) \times \mathbb{R}_{\geq 0})^*$ such that $(i_1, o_1)(i_2, o_2) \dots (i_l, o_l)$ is the sequence of input/output pairs occurring in the execution and $d_1 d_2 \dots d_l \in \mathbb{R}^l_{\geq 0}$ is a sequence of non-negative real numbers and t_k is the amount of the time elapsed since the occurrence of i_{k-1} or the beginning of the execution if no input has occurred. l is smaller than the number of steps in the execution. The *timed input sequence* and the *timed output sequence* of the execution e are $inp(e) = i_1 d_1 i_2 d_2 \dots i_l d_l$ and $out(e) = o_1 d_1 o_2 d_2 \dots o_l d_l$, respectively; they are said to be applicable and produced in (s, x), respectively. We denote by $inp(s, x)$ the set of timed input sequences applicable in (s, x). Given a timed input sequence α, let $out_{\mathcal{S}}((s, x), \alpha)$ denote the set of all timed output sequences which can be produced by $\mathcal{S}$ when α is applied in s, i.e., $out((s, x), \alpha) = \{out(e) \mid e \text{ is an execution of } \mathcal{S} \text{ in } (s, x) \text{ and } inp(e) = \alpha\}$. We let $Exec_{\mathcal{S}}$ and $Exec_{\mathcal{S}}(\alpha)$ denote the set of executions of $\mathcal{S}$ and the set of executions with the timed input sequence α.

Transitions starting in the same state are called *compatible* if they are timeout transitions or have the same input and the intersection of their timed guards is non-empty. A TFSM-TG $\mathcal{S}$ is *deterministic* (DTFSM-TG) if it has no compatible transition; otherwise, it is *non-deterministic*. $\mathcal{S}$ is *initially connected* if every state of $\mathcal{S}$ is part of a reachable timed state. Let $\lambda_{\mathcal{S}}(s,i)$ denote the set of input/output transitions defined in state s with input i. $\mathcal{S}$ is *complete* if the union of the timed guards of the transitions in $\lambda_{\mathcal{S}(s,i)}$ equals $[0,\infty[$ for every $(s,i) \in S \times I$; it implies that every $i \in I$ is the input of at least one input/output step from every reachable timed state of $\mathcal{S}$. Note that $inp(s,x) = (I \times \mathbb{R}_{\geq 0})^*$ for every timed state (s,x) of a complete machine $\mathcal{S}$.

We define distinguishability and equivalence relations between timed states of complete TFSMs-TG. Then, we extend these relations to TFSM-TG. Similar notions were introduced in [28]. Intuitively, timed states producing different timed output sequences in response to the same timed input sequence are distinguishable. Formally, let (s,x_s) and (m,x_m) be the timed states of two complete TFSMs-TG defined on the same input and output sets. Given a timed input sequence α, (s,x_s) and (m,x_m) are *distinguishable* with α denoted $(s,x_s) \not\simeq_\alpha (m,x_m)$, if the sets of timed output sequences in $out((s,x_s),\alpha)$ and $out((m,x_m),\alpha)$ differ; otherwise they are *equivalent* and we write $(s,x_s) \simeq (m,x_m)$, i.e., if the sets of timed output sequences coincide for every timed input sequence α. Two TFSM-TG are equivalent if their initial timed states are indistinguishable; otherwise they are distinguishable with a timed input sequence.

Henceforth the TFSMs-TG are complete and initially connected.

2.2 Mutants and fault model

Let $\mathcal{S} = (S, s_0, I, O, \lambda_{\mathcal{S}}, \Delta_{\mathcal{S}})$ be a complete DTFSM-TG, called the *specification* machine. A mutant is a variant of the specification and represents a possibly faulty implementation that should be detected by tests; it is also a deterministic and complete TFSM-TG. A mutant can have fewer or more states than the specification, up to a specified bound. The faults can be introduced by performing combinations of the following atomic mutation operations: changing the target state of a transition (transfer fault), changing the output of a transition (output fault), changing a timeout, merging the timed guards of transitions, splitting the timed guard of a transition, adding an extra-state. Similar operations were defined in [27,20] for testing other types of timed machines. Our operations are adapted to TFSM-TG. We compactly represent mutants with a mutation machine. Intuitively, mutants and the specification are all sub-machines of a global mutation machine describing the fault model.

TFSM-TG $\mathcal{S} = (S, s_0, I, O, \lambda_{\mathcal{S}}, \Delta_{\mathcal{S}})$ is a *sub-machine* of TFSM-TG $\mathcal{M} = (M, m_0, I, O, \lambda_{\mathcal{M}}, \Delta_{\mathcal{M}})$ if $S \subseteq M$, $s_0 = m_0$, $\lambda_{\mathcal{S}} \subseteq \lambda_{\mathcal{M}}$ and $\Delta_{\mathcal{S}} \subseteq \Delta_{\mathcal{M}}$.

Definition 2. *A non-deterministic TFSM-TG $\mathcal{M} = (M, m_0, I, O, \lambda_{\mathcal{M}}, \Delta_{\mathcal{M}})$ is a* mutation machine *of $\mathcal{S}$ if $\mathcal{S}$ is a sub-machine of $\mathcal{M}$. Transitions in $\lambda_{\mathcal{M}}$ but not in $\lambda_{\mathcal{S}}$ or in $\Delta_{\mathcal{M}}$ but not in $\Delta_{\mathcal{S}}$ are called* mutated*.*

A *mutant* is a deterministic and complete sub-machine of a mutation machine $\mathcal{M}$ different from the specification. We let $Mut(\mathcal{M})$ denote the set of mutants in $\mathcal{M}$. Let $\Delta_{\mathcal{M}}(m)$ denote the set of timeout transitions defined in m. For a state m, every mutant defines exactly one timeout transition of $\Delta_{\mathcal{M}}$ and a subset z_{mi} of transitions of $\lambda_{\mathcal{M}}(m,i)$, for every $(m,i) \in M \times I$. The subset z_{mi} satisfies the following three *cluster conditions*: (1) it is non-empty, (2) the transitions in z_{mi} are not compatible with each other and (3) the union of their timed guards equals $[0,\infty[$. We let $Z_{mi} = \{z^1_{mi}, z^2_{mi}, \ldots, z^l_{mi}\}$ be the maximal set of subsets of $\lambda_{\mathcal{M}}(m,i)$ such that z^k_{mi} satisfies the three cluster conditions, for each $k = 1...l$. The number of mutants is $|Mut(\mathcal{M})| = \prod_{(m,i)\in M\times I} |Z_{mi}| \times \prod_{m\in M} |\Delta_{\mathcal{M}}(m)| - 1$.

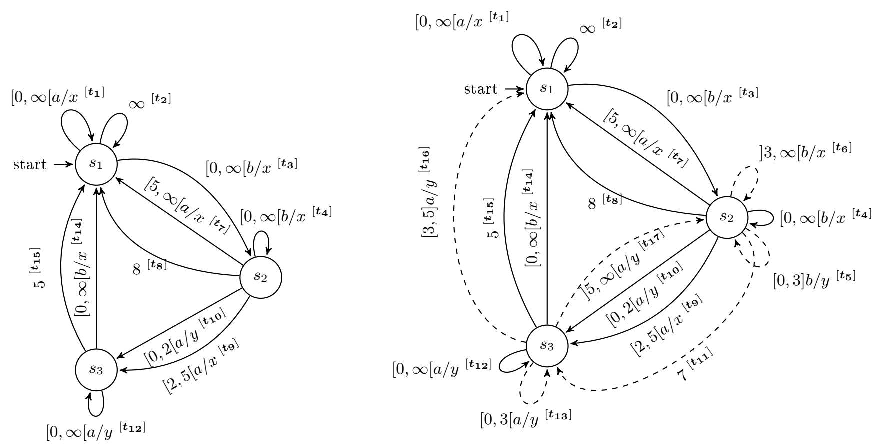
Faults in mutants are represented with mutated transitions which can be viewed as alternatives for transitions of the specification. Note that changes of delays to fire input/output transitions cannot be expressed if the specification is a TFSM with timeouts only; this is because the timed guard for every input in a TFSM with timeouts is always $[0,\infty[$. Some executions of non-deterministic sub-machines of $\mathcal{M}$ are not executions of any mutants or the specification. We can show that $\bigcup_{P\in Mut(\mathcal{M})\cup\{\mathcal{S}\}} Exec_{\mathcal{P}}(\alpha) \subseteq Exec_{\mathcal{M}}(\alpha)$.

A transition t is *suspicious* in $\mathcal{M}$ if $\mathcal{M}$ defines another transition t' compatible with t. In other words, t and t' specify possible behaviors of different mutants. A transition of the specification is called *untrusted* if it is suspicious in the mutation machine; otherwise, it is *trusted*. Every trusted transition is defined in each mutant. The set of suspicious transitions of $\mathcal{M}$ is partitioned into a set of untrusted transitions all defined in the specification and the set of mutated transitions undefined in the specification. We let $Susp_O$ denote the set of suspicious transitions in an artifact O containing transitions.

Figure 1b presents a mutation machine $\mathcal{M}_1$. Transitions represented with dashed lines are mutated. Transition identifiers appear in brackets. $\mathcal{M}_1$ is non-deterministic because, e.g., t_4 and t_6 are compatible or the two timeout transitions t_8 and t_{11} start from s_2. The suspicious transitions in $Susp_{\mathcal{M}_1}$ are $t_4, t_5, t_6, t_8, t_{11}, t_{12}, t_{13}, t_{16}, t_{17}$; the other transitions are trusted. The specification $\mathcal{S}_1$ in Figure 1a is deterministic; it defines all the trusted transitions and the untrusted transitions t_4, t_8, t_{12}. $\Delta_{\mathcal{M}_1}(s_1) = \{t_2\}$, $Z_{s_1a} = \{\{t_1\}\}$, $Z_{s_1b} = \{\{t_3\}\}$, $\Delta_{\mathcal{M}_1}(s_2) = \{t_8, t_{11}\}$, $Z_{s_2a} = \{\{t_7, t_9, t_{10}\}\}$, $Z_{s_2b} = \{\{t_4\}, \{t_5, t_6\}\}$, $\Delta_{\mathcal{M}_3}(s_1) = \{t_{15}\}$, $Z_{s_3a} = \{\{t_{12}\}, \{t_{13}, t_{16}, t_{17}\}\}$ and $Z_{s_3b} = \{\{t_{14}\}\}$. $Mut(\mathcal{M}_1)$ contains seven mutants; two of them appear in Figure 2a and Figure 2b. The mutants are obtained from the specification by changing the behavior of suspicious transitions, which is done by replacing the untrusted transitions in the specification by other suspicious transitions they are compatible with. The mutant in Figure 2b has a different behavior in state s_3 for input a. The mutants and the specification have the same behavior in s_1 for input a; this is because t_1 is trusted.

Let $\mathcal{P}$ be a mutant with an initial state p_0 of the mutation machine $\mathcal{M}$ of $\mathcal{S}$. We use the equivalence relation $\simeq$ to define conforming mutants.

Definition 3. *Mutant $\mathcal{P}$* conforms *to $\mathcal{S}$, if $(p_0, 0) \simeq (s_0, 0)$; otherwise, it is nonconforming and a timed input sequence α such that $(p_0, 0) \not\simeq_\alpha (s_0, 0)$ is said to* detect *$\mathcal{P}$. $\mathcal{P}$* survives *α if α does not detect $\mathcal{P}$.*



(a) The specification machine $\mathcal{S}_1$ (b) The mutation machine $\mathcal{M}_1$

Fig. 1: A mutation machine for a specification machine; s_1 is the initial state

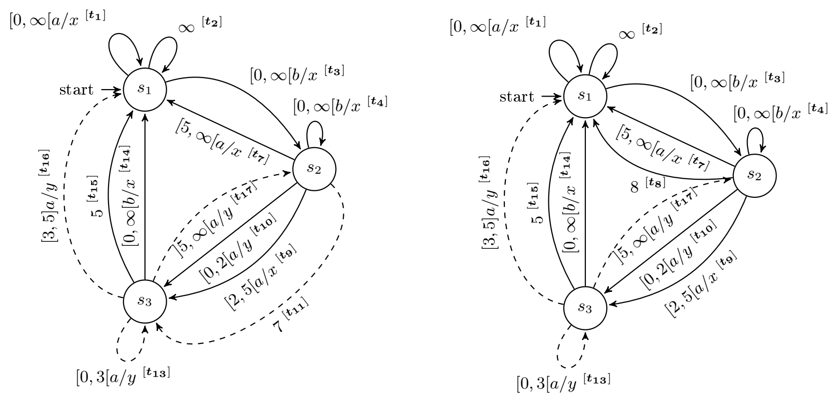
The set $Mut(\mathcal{M})$ of all mutants in mutation machine $\mathcal{M}$ is called a *fault domain* for $\mathcal{S}$. If $\mathcal{M}$ is deterministic and complete then $\mathcal{M}$ includes only the specification and $Mut(\mathcal{M})$ is empty. A general *fault model* is the tuple $\langle \mathcal{S}, \simeq, Mut(\mathcal{M}) \rangle$ following [22,18]. The conformance relation partitions the set $Mut(\mathcal{M})$ into conforming mutants and nonconforming ones which we need to detect.

Definition 4. *A* test *for $\langle \mathcal{S}, \simeq, Mut(\mathcal{M}) \rangle$ is a timed input sequence. A* complete test suite *for $\langle \mathcal{S}, \simeq, Mut(\mathcal{M}) \rangle$ is a set of tests detecting all nonconforming mutants in $Mut(\mathcal{M})$.*

We address the problem of checking the completeness of a test suite and the problem of generating a complete test suite for a fault model $\langle \mathcal{S}, \simeq, Mut(\mathcal{M}) \rangle$, where the specification machine $\mathcal{S}$ is deterministic and complete and the mutation machine can have more states than $\mathcal{S}$. Our approach consists in eliminating, from the fault model, the mutants detected by tests meanwhile avoiding their one-by-one enumeration. Undetected mutants will serve to generate new tests.

3 Revealing combs for characterizing detected mutants

We introduce combs to characterize mutants having common executions of the mutation machine since mutation machine includes all the mutants. The mutants defining all the transitions in a comb have common executions and can be detected with the same test. The comb for an execution e is the sequence of transitions permitting the steps in e; it is denoted by π_e. The transitions in combs do not necessarily form paths in the state-transition diagram of $\mathcal{M}$. This is because they contain non-fired time-out transitions permitting time-elapsing steps when



(a) Nonconforming mutant $\mathcal{P}_1$ (b) Nonconforming mutant $\mathcal{P}_2$

Fig. 2: Two mutants included in the mutation machine $\mathcal{M}_1$ in Figure 1b

timeouts are not expired. Without the time-elapsing steps, input-output steps would not be possible because a certain delay is needed for reaching a timed guard; delays are possible up to the expiration of maximal timeouts. Fired timeout transitions also occur in combs. Let $\Pi_{\mathcal{Q}}(\alpha) = \{\pi_e | e \in Exec_{\mathcal{Q}}(\alpha)\}$ denote the set of combs for the executions of $\mathcal{Q}$ with timed input sequence α and $\Pi_{\mathcal{Q}}$ denote the set of combs for the executions of a TFSM-TG $\mathcal{Q}$. It holds that $\bigcup_{\mathcal{P} \in Mut(\mathcal{M}) \cup \{\mathcal{S}\}} \Pi_{\mathcal{P}}(\alpha) \subseteq \Pi_{\mathcal{M}}(\alpha)$ and $\bigcup_{\mathcal{P} \in Mut(\mathcal{M}) \cup \{\mathcal{S}\}} \Pi_{\mathcal{P}} \subseteq \Pi_{\mathcal{M}}$. An execution $e \in Exec_{\mathcal{M}}(\alpha)$ is called *unexpected* if there is $e' \in Exec_{\mathcal{S}}(\alpha)$ such that $out(e') \neq out(e)$; otherwise it is expected and $out(e)$ is called expected.

Definition 5. *A comb is* deterministic *if its transitions are pairwise non-compatible. A deterministic comb is* revealing *if it corresponds to an unexpected execution of the mutation machine.*

The executions of the mutants correspond to deterministic combs. Only executions of a non-deterministic sub-machine of $\mathcal{M}$ can correspond to a non-deterministic comb. Mutants included in $\mathcal{M}$ can have common executions. Because we want to avoid the enumeration of the mutants one-by-one, we would like to identify the mutants which have an execution in common, i.e., they are *involved* in the execution. A mutant $\mathcal{P}$ is involved in an execution e of $\mathcal{M}$ if $\mathcal{P}$ defines all the suspicious transitions in the comb π_e of e, i.e., $Susp_{\pi_e} \subseteq \lambda_{\mathcal{P}} \cup \Delta_{\mathcal{P}}$. This is because every mutant defines all the trusted transitions in $\mathcal{M}$ and only suspicious transitions vary between the mutants. We let $Rev_{\mathcal{M}}(\alpha)$ denote the set of revealing combs for executions with timed input sequence α. It holds that $Rev_{\mathcal{M}}(\alpha) = \bigcup_{\mathcal{P} \in Mut(\mathcal{M})} Rev_{\mathcal{P}}(\alpha)$. The following lemma indicates that detected mutants are involved in revealing combs of the mutation machine.

Lemma 1. *$Mut(\mathcal{M})$ contains nonconforming mutants if and only if $Rev_{\mathcal{M}}(\alpha) \neq \emptyset$, for some timed input sequence α.*

Lemma 2. *Test α detects mutant $\mathcal{P}$ if and only if there exists $e \in Exec_{\mathcal{M}}(\alpha)$ such that $\pi_e \in Rev_{\mathcal{M}}(\alpha)$ and $Susp_{\pi_e} \subseteq \lambda_{\mathcal{P}} \cup \Delta_{\mathcal{P}}$.*

Corollary 1. *Mutant $\mathcal{P}$ survives α if and only if for every $\pi \in Rev_{\mathcal{M}}(\alpha)$, some transitions in $Susp_{\pi}$ do not belong to $\lambda_{\mathcal{P}} \cup \Delta_{\mathcal{P}}$.*

$e_1 = (s_1, 0)0.5^{[t_2]}(s_1, 0.5)b/x^{[t_3]}(s_2, 0)7^{[t_8]}(s_2, 7)0.5^{[t_8]}(s_2, 7.5)a/x^{[t_7]}(s_1, 0)$ is an execution of $\mathcal{M}_1$ in Figure 1b; it is also an execution of $\mathcal{S}_1$. The transitions permitting the steps appear in brackets and also serve to identify the comb. The first, third and fourth steps in e_1 are time-elapsing. t_8 permits the fourth step. A distinct execution, let us call it e_2, could involve t_{11} at the fourth step. Another distinct execution is possible by making a timeout step at the third step, i.e., by firing t_{11}. The comb $\pi_{e_1} = t_2 \curvearrowright t_3 t_8 t_8 \curvearrowright t_7$ is for e_1 does not form a path in $\mathcal{M}_1$; the symbol "$\curvearrowright$" is inserted between timeout transitions permitting time-elapsing steps which just precede input/output steps. π_{e_1} is deterministic and a mutant can define all the suspicious transitions in it. π_{e_1} is not revealing because the timed output sequence of e_1, $x0.5x7.5$ is expected. The mutants defining only the suspicious transitions occurring in π_{e_1} and all the trusted transitions cannot be detected by $b0.5a7.5$. $\pi_{e_2} = t_2 \curvearrowright t_3 t_8 t_{11} \curvearrowright t_7$ is not deterministic because t_8 and t_{11} are compatible and suspicious; a mutant cannot define these two transitions. π_{e_2} is not revealing since it is not deterministic.

4 Distinguishing Automaton and Revealing Combs

The distinguishing automaton for a specification $\mathcal{S}$ and a mutation machine $\mathcal{M}$ is aimed to represent synchronous executions of $\mathcal{S}$ and the mutants in $\mathcal{M}$. It will be used to identify unexpected executions of the mutants and the corresponding combs without enumerating the mutants one-by-one. Each of its states is composed of a timed state of $\mathcal{S}$ and a timed state of $\mathcal{M}$. Each transition between two states of the automaton represents a synchronization between a transition of $\mathcal{S}$ and a transition of $\mathcal{M}$. Because $\mathcal{S}$ and $\mathcal{M}$ could have an infinite number of reachable timed states, a naive definition of the distinguishing automaton could include an infinite number of the automaton's states. For this reason our definition is based on a new notion of abstract timed states for TFSM-TG.

4.1 Abstract timed state for TFSM-TG

We introduce a new notion of abstract execution which we show to be equivalent to the notion of execution described in Section 2. In abstract executions the range for clock values is finite while it is infinite in executions. In particular, by increasing a clock value by one in a state which defines the infinite timeout, the value of the clock will continuously increase up to a value which depends on the number of time-elapsing steps. It happens that after a certain threshold value, the exact value of the clock is not necessary to determine transitions which can be fired. In state m, the threshold value is the maximal finite integer used in input-output and timeout transitions starting at state m, denoted by

max_m. We let max_m^+ be a special number representing any real number greater than max_m, the maximal finite value in transitions starting from m; it has the following arithmetic properties: $max_m^+ \in]max_m, \infty[$, $max_m^+ + k = max_m^+$ for every finite real number $k \geq 0$ and $max_m^+ + \infty = \infty$.

The abstract time domain for the clock when the TFSM-TG is in state m is $Atd_m = [0, max_m] \cup \{max_m^+\}$ where $[0, max_m]$ is an integer interval. For machine with states in M, the abstract time domain is $Atd_S = \bigcup_{m \in S} Atd_m$. An abstract execution of a TFSM-TG is defined with abstract execution steps which only consider abstract time domains for clocks and arithmetic properties introduced for the domain. Every execution corresponds to an abstract execution. The latter can be obtained from the former by replacing every timed state (s, x) with (s, max_m^+) wherever x is greater than max_m. We can show that there is an abstract execution of a TFSM-TG with timed input/output sequence α/β if and only if there is an execution of the TFSM-TG with α/β. This indicates that abstract executions can be used instead of executions.

4.2 Distinguishing automaton

Definition 6. *Given a specification machine $\mathcal{S} = (S, s_0, I, O, \lambda_{\mathcal{S}}, \Delta_{\mathcal{S}})$ and a mutation machine $\mathcal{M} = (M, m_0, I, O, \lambda_{\mathcal{M}}, \Delta_{\mathcal{M}})$, a finite automaton $\mathcal{D} = (C \cup \{\nabla\}, c_0, I, \lambda_{\mathcal{D}}, \Delta_{\mathcal{D}}, \nabla)$, where $C \subseteq S \times S \times Atd_S \times Atd_M$, $\lambda_{\mathcal{D}} \subseteq C \times I \times \Pi \times C$ is the input transition relation, $\Delta_{\mathcal{D}} \subseteq C \times (\mathbb{N}_{\geq 1} \cup \{\infty\}) \times C$ is the timeout transition relation and ∇ is the accepting (sink) state, is the* distinguishing automaton with timeouts and timed guards *for $\mathcal{S}$ and $\mathcal{M}$, if it holds that:*

- $c_0 = (s_0, m_0, 0, 0)$
- *For each $(s, m, x_s, x_m) \in C$ and $i \in I$*
  - $(\mathcal{R}_1)$: $((s, m, x_s, x_m), i, \pi_s^{-x_s} \cap \pi_m^{-x_m}, (s', m', 0, 0)) \in \lambda_{\mathcal{D}}$ *if there exists* $(s, i, \pi_s, o_s, s') \in \lambda_{\mathcal{S}}, (m, i, \pi_m, o_m, m') \in \lambda_{\mathcal{M}}$ *s.t.* $\pi_s^{-x_s} \cap \pi_m^{-x_m} \neq \emptyset$ *and* $o_s = o_m$
  - $(\mathcal{R}_2)$: $((s, m, x_s, x_m), i, \pi_s^{-x_s} \cap \pi_m^{-x_m}, \nabla) \in \lambda_{\mathcal{D}}$ *if there exists* $(s, i, \pi_s, o_s, s') \in \lambda_{\mathcal{S}}, (m, i, \pi_m, o_m, m') \in \lambda_{\mathcal{M}}$ *s.t.* $\pi_s^{-x_s} \cap \pi_m^{-x_m} \neq \emptyset$ *and* $o_s \neq o_m$
- *For each $(s, m, x_s, x_m) \in C$ and the only timeout transition $(s, \delta_s, s') \in \Delta_{\mathcal{S}}$ defined in the state of the deterministic specification*
  - $(\mathcal{R}_3)$: $((s, m, x_s, x_m), \delta_m - x_m, (s', m', 0, 0)) \in \Delta_{\mathcal{D}}$ *if there exists* $(m, \delta_m, m') \in \Delta_{\mathcal{M}}$ *s.t.* $\delta_s - x_s = \delta_m - x_m$ *and* $\delta_m - x_m > 0$
  - $(\mathcal{R}_4)$: $((s, m, x_s, x_m), \delta_m - x_m, (s, m', max_s^+, 0)) \in \Delta_{\mathcal{D}}$ *if there exists* $(m, \delta_m, m') \in \Delta_{\mathcal{M}}$ *such that* $\delta_m - x_m > 0$, $\delta_s - x_s > \delta_m - x_m$, *and* $x_s + \delta_m - x_m > max_s$
  - $(\mathcal{R}_5)$: $((s, m, x_s, x_m), \delta_m - x_m, (s, m', x_s + \delta_m - x_m, 0)) \in \Delta_{\mathcal{D}}$ *if there exists* $(m, \delta_m, m') \in \Delta_{\mathcal{M}}$ *such that* $\delta_m - x_m > 0$, $\delta_s - x_s > \delta_m - x_m$, *and* $x_s + \delta_m - x_m \leq max_s$
  - $(\mathcal{R}_6)$: $((s, m, x_s, x_m), \delta_s - x_s, (s', m, 0, max_m^+)) \in \Delta_{\mathcal{D}}$ *if there exists* $(m, \delta_m, m') \in \Delta_{\mathcal{M}}$ *such that* $\delta_s - x_s > 0$, $\delta_s - x_s < \delta_m - x_m$, *and* $x_m + \delta_s - x_s > max_m$

($\mathcal{R}_7$) : $((s, m, x_s, x_m), \delta_s - x_s, (s', m, 0, x_m + \delta_s - x_s)) \in \Delta_{\mathcal{D}}$ *if there exists* $(m, \delta_m, m') \in \Delta_{\mathcal{M}}$ *such that* $\delta_s - x_s > 0$*,* $\delta_s - x_s < \delta_m - x_m$*, and* $x_m + \delta_s - x_s \leq max_m$

– $(\nabla, i, [0, \infty[, \nabla) \in \lambda_{\mathcal{D}}$ *for all* $i \in I$ *and* $(\nabla, \infty, \nabla) \in \Delta_{\mathcal{D}}$

In a state (s, m, x_s, s_m) of $\mathcal{D}$, (s, x_s) and (m, s_m) represents timed state of the specification and mutation machines. The two machines synchronize on input actions (see $\mathcal{R}_1$, $\mathcal{R}_2$) and time delays. They do not synchronize on timeout, i.e., the expiration of a timeout in a machine is not necessarily synchronized with the expiration of a timeout in the other machine (case of $\mathcal{R}_4$ to $\mathcal{R}_7$); A synchronization of timeout transitions may occur depending on the respective clocks (see $\mathcal{R}_4$). The automaton also uses a sink state ∇ to identify synchronizations on an input but with different outputs. An execution of $\mathcal{D}$ from a timed state (c, x) is a sequence of steps between timed states of $\mathcal{D}$; it can be defined similarly to that for a TFSM-TG. An execution starting from $(c_0, 0)$ and ending at ∇ is called *accepted*. $\mathcal{D}$ is complete because $\mathcal{S}$ and $\mathcal{M}$ are complete. Every execution of $\mathcal{D}$ corresponds to an execution of the specification $\mathcal{S}$ and an execution of the mutation machine $\mathcal{M}$. Indeed, transitions in $\mathcal{D}$ are directly defined by execution steps of $\mathcal{S}$ and $\mathcal{M}$ and indirectly by transitions of $\mathcal{S}$ and $\mathcal{M}$ permitting the steps. We let $e = (e_1, e_2)$ represent an execution of $\mathcal{D}$, where e_1 and e_2 are the corresponding executions of $\mathcal{S}$ and $\mathcal{M}$. Clearly $inp(e) = inp(e_1) = inp(e_2)$. For every execution e_1 of $\mathcal{S}$ there is an execution e_2 of $\mathcal{M}$ such that $e = (e_1, e_2)$ is an execution of $\mathcal{D}$. For every execution e_2 of $\mathcal{M}$ there is an execution e_1 of $\mathcal{S}$ such that $e = (e_1, e_2)$ is an execution of $\mathcal{D}$. Moreover e_1 and e'_1 have the same timed input/output sequence whenever (e_1, e_2) and (e'_1, e_2) represent executions of $\mathcal{D}$. It means that $\mathcal{D}$ represents the comparisons of timed input/output sequences of $\mathcal{M}$ with an expected timed input/output sequence of $\mathcal{S}$.

Lemma 3. *An execution* e_2 *of* $\mathcal{M}$ *is unexpected if and only if there is an execution* e_1 *of* $\mathcal{S}$ *such that* $e = (e_1, e_2)$ *is an accepted execution of* $\mathcal{D}$*.*

Only unexpected executions of $\mathcal{M}$ having deterministic combs can be executions of mutants; the combs for such executions are revealing; they could be identified in checking whether $Mut(\mathcal{M})$ contains nonconforming mutants.

Lemma 4. $Rev_{\mathcal{M}}(\alpha) \neq \emptyset$ *if and only if there exists an accepted execution* $e = (e_1, e_2)$ *of* $\mathcal{D}$ *such that* π_{e_2} *is a deterministic comb and* $\alpha = inp(e_2)$*.*

The deterministic revealing combs for a test α can be computed from the distinguishing automaton for the specification and mutation machines. Their computation works as follows. First we compute the accepted executions of the distinguishing automaton $\mathcal{D}$ with α; this can be done by defining a product an automaton for α and $\mathcal{D}$, which we do not formalize for the sake of simplicity. Each accepted execution corresponds to an execution of the specification and an unexpected execution of the mutation machine; the deterministic combs for the unexpected execution of the mutation machine belong to $Rev_{\mathcal{M}}(\alpha)$.

The following corollary is a consequence of Lemma 4 and Lemma 1.

Corollary 2. *$Mut(\mathcal{M})$ contains nonconforming mutants detectable with α if and only if there exists an accepted execution $e = (e_1, e_2)$ of $\mathcal{D}$ such that π_{e_2} is a deterministic comb and $\alpha = inp(e_2)$.*

5 Boolean Formulas Encoding (Un)Detected Mutants

We encode the test-surviving mutants with Boolean formulas over Boolean variables; each variable corresponds to a transition in $\mathcal{M}$. Henceforth we let variable t represent both a transition in $\mathcal{M}$ and the corresponding variable. A solution of a Boolean formula assigns *True* or *False* to the variables, which we can use to choose a (mutant) sub-machine in $\mathcal{M}$. Intuitively, we can replace in a mutant a suspicious transition by another (compatible) suspicious transition to obtain a different mutant. However, every mutant defines all the trusted transitions. We say that a sub-machine (possibly a mutant) of $\mathcal{M}$ is *determined* by a Boolean formula φ defined over the transition variables of $\mathcal{M}$ if: (1) the sub-machine includes all the trusted transitions in $\mathcal{M}$ and, (2) there exists a solution of φ which assigns *True* to a transition variable if and only if the corresponding transition is in the sub-machine. The encoding Boolean formula is the conjunction of two sub-formulas. The first sub-formula encodes the complementary of the detected mutants, which can contain mutants and non-deterministic sub-machines of the mutation machine as well. The second sub-formula encodes all the mutants only.

The encoding of the detected mutants uses the suspicious transitions in revealing combs corresponding to the tests in a test suites. This encoding is inspired by Lemma 2. Given a set of revealing combs $Rev_{\mathcal{M}}(\alpha)$ for test α, we let $\varphi_\alpha = \bigvee_{\pi \in Rev_{\mathcal{M}}(\alpha)} \left(\bigwedge_{t \in Susp_\pi} t \right)$ be a Boolean formula over the variables for the suspicious transitions in revealing combs of $Rev_{\mathcal{M}}(\alpha)$. φ_α encodes all the mutants and also non-deterministic sub-machines involved in revealing combs; this means that every solution of φ_α determines a sub-machine of $\mathcal{M}$ that defines all the transitions in a comb $\pi \in Rev_{\mathcal{M}}(\alpha)$. So, φ_α determines all the sub-machines with unexpected outputs for input α. Let $\neg\varphi$ denote the negation of φ. Every solution of $\neg\varphi_\alpha$ sets variables for some suspicious transitions in every revealing comb to *False*; this indicates that every mutant determined by $\neg\varphi_\alpha$ does not define some suspicious transitions from each comb in $Rev_{\mathcal{M}}(\alpha)$.

Lemma 5. *For every $\mathcal{P}$ determined by $\neg\varphi_\alpha$ and every $\pi \in Rev_{\mathcal{M}}(\alpha)$, there exists $t \in Susp_\pi$ which does not belong to $\lambda_{\mathcal{P}} \cup \Delta_{\mathcal{P}}$.*

The following lemma is a consequence of Lemma 5 and Corollary 1.

Lemma 6. *Every mutant determined by $\neg\varphi_\alpha$ survives α.*

Let $TS = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ be a test suite. We define $\varphi_{TS} = \bigvee_{\alpha_i \in TS} \varphi_{\alpha_i}$. The formula φ_{TS} determines all the sub-machines which produce unexpected outputs for an input $\alpha_i \in TS$, i.e., the sub-machines detected by at least one test in TS. A determined sub-machine is not necessarily a mutant; so we need to encode all the mutants in $\mathcal{M}$. We can proof the following lemma by using Lemma 6.

Lemma 7. *Every mutant determined by $\neg\varphi_{TS}$ survives the test suite TS.*

Note that $\neg\varphi_{TS}$ determines not only mutants but also non-deterministic sub-machines of the mutation machine $\mathcal{M} = (M, m_0, I, O, \lambda_{\mathcal{M}}, \Delta_{\mathcal{M}})$. In other to exclude these non-deterministic sub-machines of $\mathcal{M}$, we encode the set of mutants with a Boolean formula $\varphi_{\mathcal{M}}$. Each of its solutions determines exactly one timeout transition in every state (which is expressed by $(Eq\ 2)$) and a subset $z_{mi} \in Z_{mi}$, where Z_{mi} is the maximal set of subsets of $\lambda_{\mathcal{M}}(m, i)$ satisfying the cluster conditions introduced in Section 2.2. Let us define:

$$\varphi_{\mathcal{M}} = \bigwedge_{m\in M}\left(\varphi_{\Delta m} \wedge \bigwedge_{i\in I}\varphi_{mi}\right) \wedge \bigvee_{t\in\lambda_{\mathcal{S}}\cup\Delta_{\mathcal{S}}} \neg t \qquad (Eq\ 1)$$

where

$$\varphi_{\Delta m} = \bigvee_{t\in\Delta_{\mathcal{M}}(m)}\left(t \wedge \bigwedge_{t'\in\Delta_{\mathcal{M}}(m), t'\neq t} \neg t'\right) \qquad (Eq\ 2)$$

$$\varphi_{mi} = \bigvee_{z\in Z_{mi}}\left(\varphi_z \wedge \bigwedge_{w\in Z_{mi}, w\neq z} \neg\varphi_w\right) \text{ and } \varphi_z = \bigwedge_{t\in z} t \qquad (Eq\ 3)$$

Each solution of $\varphi_{\Delta m}$ sets to *True* the variable of exactly one timeout transition defined in m. All the variables in exactly one set Z_{mi} has all its variables assigned to *True* by each solution of φ_{mi}. The variable for each trusted transition is set to $True$ in every solution of $\varphi_{m\delta}$; this is because they are not compatible with any other transitions and thus defined by every mutant. We can conclude that each solution of $\varphi_{\mathcal{M}}$ determines a mutant in $Mut(\mathcal{M})$ and every mutant is determined by a solution of $\varphi_{\mathcal{M}}$, i.e., $\varphi_{\mathcal{M}}$ determines all the mutants in $Mut(\mathcal{M})$. According to its definition, every solution of $\varphi_{\mathcal{M}}$ never assigns *True* to the variables for compatible transitions and determines a mutant.

Lemma 8. *$\varphi_{\mathcal{M}}$ determines the mutants in $Mut(\mathcal{M})$.*

We can prove the following theorem by using Lemma 8 and Lemma 7.

Theorem 1. *Formula $\varphi_{\mathcal{M}} \wedge \neg\varphi_{TS}$ determines exactly the mutants undetected by the test suite TS.*

For $\mathcal{M}_1$ in Figure 1b, $\varphi_{\Delta s_2} = (t_8 \wedge \neg t_{11}) \vee (t_{11} \wedge \neg t_8)$, $\varphi_{s_2 a} = t_7 \wedge t_9 \wedge t_{10}$ and $\varphi_{s2b} = (t_4 \wedge \neg t_5 \wedge \neg t_6) \vee (t_5 \wedge t_6 \wedge \neg t_4)$. $\varphi_{\Delta s_2} \wedge \varphi_{s_2 a} \wedge \varphi_{s_2 b}$ determines the transitions starting at s_2 in a mutant. We can make similar formulas for s_1 and s_3, according to the sets $Z_{s_1 a}, Z_{s_1 b}, Z_{s_2 a}, Z_{s_2 b}, Z_{s_3 a}$ and $Z_{s_3 b}$ presented before Definition 3. Finally, $\varphi_{\mathcal{M}_1} = \varphi_{\Delta s_1} \wedge \varphi_{s_1 a} \wedge \varphi_{s_1 b} \wedge \varphi_{\Delta s_2} \wedge \varphi_{s_2 a} \wedge \varphi_{s_2 b} \wedge \varphi_{\Delta s_3} \wedge \varphi_{s_3 a} \wedge \varphi_{s_3 b}$.

The test $\alpha_1 = b0.5a7.5$ triggers executions in the distinguishing automaton for $\mathcal{M}_1$. These executions correspond to the expected execution e_1, e_2 and :
$e_3 = (s_1, 0)0.5^{[t_2]}(s_1, 0.5)b/x^{[t_3]}(s_2, 0)7^{[t_{11}]}(s_3, 0)0.5^{[t_{15}]}(s_3, 0.5)a/y^{[t_{12}]}(s_1, 0)$ and
$e_4 = (s_1, 0)0.5^{[t_2]}(s_1, 0.5)b/x^{[t_3]}(s_2, 0)7^{[t_{11}]}(s_3, 0)0.5^{[t_{15}]}(s_3, 0.5)a/y^{[t_{13}]}(s_1, 0)$. The comb for e_3 and e_4 are $\pi_{e_3} = t_2 \curvearrowright t_3 t_{11} t_{15} \curvearrowright t_{12}$ and $\pi_{e_4} = t_2 \curvearrowright t_3 t_{11} t_{15} \curvearrowright t_{13}$; they are revealing because the produced timed output sequence $x0.5y7.5$ is unexpected; so they characterize the mutants detected by test α_1. We recall that π_{e_1} and π_{e_2} are not revealing. The formula $\neg\varphi_{\alpha_1} = \neg((t_{11} \wedge t_{12}) \vee (t_{11} \wedge t_{13}))$ encodes the mutants surviving α_1, e,g., the mutant $\mathcal{P}_2$ in Figure 2b.

6 Test Analysis and Test Generation

Consider a test suite TS and a fault model $\langle \mathcal{M}, \simeq, Mut(\mathcal{M})\rangle$. The test analysis problem consists in verifying whether the test suite is complete for the fault model. The test generation problem aims at generating a complete test suite, by adding new tests to the given one. Solving the test analysis problem, we encode the mutants undetected by the tests with the Boolean formula $\varphi_{\mathcal{M}} \wedge \neg\varphi_{TS}$; then we use a solver to determine a mutant undetected by TS, according to Theorem 1 TS is complete if no mutant can be determined or only conforming mutants can be determined. We exclude encountered conforming mutants from the fault domain; this is done by making the conjunction of $\varphi_{\mathcal{M}} \wedge \neg\varphi_{TS}$ with the formula $\bigvee_{t\in\lambda_{\mathcal{P}}\cup\Delta_{\mathcal{P}}} \neg t$ encoding the mutants different from $\mathcal{P}$.

Theorem 2. *TS is complete for $\langle \mathcal{M}, \simeq, Mut(\mathcal{M})\rangle$ if and only if $\varphi_{\mathcal{M}} \wedge \neg\varphi_{TS}$ is not satisfiable or all the mutants it determines are conforming.*

In order to generate a complete test suite, our procedure analyzes the current test suite TS. If it is complete the procedure stops and return TS; otherwise we generate a test α detecting nonconforming mutants from a mutant surviving TS. α can be the timed input sequence of an execution of the distinguishing automaton for the specification and a surviving mutant (which is a part of the mutation machine), according to Corollary 4. Then we compute φ_α and we analyze the new test suite $TS' = TS \cup \{\alpha\}$ by solving the formula $\varphi_{\mathcal{M}} \wedge \neg\varphi_{TS} \wedge \neg\varphi_\alpha$; this may trigger the generation of a new test. We generate iteratively new tests until all the procedure stops and a complete test suite is returned.

For example, the test suite $TS = \{b0.5a7.5\}$ is not complete for $\mathcal{M}_1$ in Figure 1b because $\varphi_{\mathcal{M}_1} \wedge \neg\varphi_{TS}$ determines the nonconforming mutant $\mathcal{P}_2$ in Figure 2b, where $\varphi_{TS} = \varphi_{\alpha_1}$ and $\alpha_1 = b0.5a7.5$. An implementation of our approach generates the complete test suite $\{\alpha_1 = b0.5a7.5, \alpha_2 = b0.5a0.5a3.5a0.5, \alpha_3 = b0.5b0.5\}$ detecting all the seven nonconforming mutants. α_2 detects $\mathcal{P}_2$; it is the timed input sequence of an accepted execution of the distinguishing automaton of $\mathcal{S}_1$ and $\mathcal{P}_2$. A nonconforming mutant undetected by α_1 and α_2 was used to generate α_3; it defines transition t_5.

Let us compare our complete test suite generation approach with the one proposed in [26]. We focus on detecting specific faults represented in the mutation machine which can have more states than the specification, so do the mutants. The specification machine is not necessarily "minimal". We believe that for the traditional fault model, the method in [26] could be faster than ours (especially if the time required to minimize the specification is not considered), but the method in [26] could generate redundant tests. The reason is that the method in [26] is an application of the W-method over a classical TFSM which represents an abstraction of TFSM-TG and the W-method was developed for minimal classical FSM. The size of the abstract TFSM-TG [26] could contribute to decrease the efficiency of the test generation [8,26]. The W-method generates tests by combining input sequences (namely, state cover sets, all input sequences of a certain length and state identification input sequences or characterization sets);

Table 1: Size of the generated complete test suites and generating time ; for an entry (x, y), x is the size of the test suite and y is the generating time in seconds

	#mutants in the fault domain		
#states	$\simeq 10^4$	$\simeq 10^8$	$\simeq 10^{12}$
8 states	(6, 0.22)	(7, 0.32)	(12, 45.57)
10 states	(3, 0.16)	(8, 2.34)	(21, 46.94)
12 states	(5, 0.72)	(5, 2.77)	(15, 4.45)

it does not check whether each generated test actually detects faults and several generated tests can detects only one mutant. Thus the number of generated tests can be bigger and less adequate to focus on faults specified in customized fault models represented with mutation machines. Where [26] will generate a huge test suite to detect all possible faults, our method will generate test suites of reduced sizes for customized fault models.

We performed an empirical evaluation of our approach with a proof-of-concept tool developed in C++ and randomly generated TFSM-TG with two inputs and outputs, with a maximal timeout of 5 (resp. 10) for the specification (resp. the mutations machines). The tool uses cryptoSAT [24]. Preliminary results appear in Table 1 which presents sizes and generating times for test suites from mutation machines defining multiple of mutants with few states. For each number of states and number of mutants, we generated tests for several mutation machines. The tool can take a long time in encoding the fault domain; this may happen when there are too many compatible (not necessarily equal) timed guards and timeouts defined in the same state. Since this operation is done for every state, it could be distributed. In most of the situations, the tool can rapidly encode the fault domain and the test generation becomes faster.

7 Conclusion

We have proposed a multiple mutation testing theory to testing real-time systems represented with finite state machine extended with timed guards and timeouts (TFSM-TG). We developed an approach to generate complete test suites for fault models represented with mutation machines. The approach relies on the definition of distinguishing automaton for mutation machine and the construction of Boolean formulas encoding the (faulty) implementations undetected by tests. We implemented the approach in a proof-of-concept tool which we used to evaluate the efficiency of the approach. The experimental results show that the approach can be used to derive tests for non-trivial TFSM-TG fault models representing an important number of faults.

Ongoing work includes developing open access benchmarks and use them to compare the existing test generation methods for TFSM-TG. We also plan to generate symbolic tests, i.e., timed input sequences with delay intervals instead of simple delays, and complete test suites consisting of a single test.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183 – 235 (1994)
2. Bertrand, N., Jéron, T., Stainer, A., Krichen, M.: Off-line test selection with test purposes for non-deterministic timed automata. In: Proceedings of TACAS'2011. pp. 96–111. Springer (2011)
3. Bohnenkamp, H., Belinfante, A.: Timed testing with TorX. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005: Formal Methods. pp. 173–188. Springer (2005)
4. Bousquet, L.D., Ouabdesselam, F., Richier, J.L., Zuanon, N.: Lutess: A specification-driven testing environment for synchronous software. In: Proceedings of ICSE'1999. pp. 267–276 (1999)
5. Bresolin, D., El-Fakih, K., Villa, T., Yevtushenko, N.: Deterministic Timed Finite State Machines: Equivalence Checking and Expressive Power. In: GandALF (2014)
6. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering SE-4(3), 178–187 (1978)
7. Derderian, K., Merayo, M.G., Hierons, R.M., Núñez, M.: Aiding test case generation in temporally constrained state based systems using genetic algorithms. In: IWANN'2009. pp. 327–334. Springer (2009)
8. Dorofeeva, R., El-Fakih, K., Maag, S., Cavalli, A.R., Yevtushenko, N.: Fsm-based conformance testing methods: A survey annotated with experimental evaluation. Information & Software Technology 52(12), 1286–1297 (2010)
9. El-Fakih, K., Yevtushenko, N., Fouchal, H.: Testing Timed Finite State Machines with Guaranteed Fault Coverage. In: Proceedings of the 21st TESTCOM and 9th FATES. pp. 66–80. Springer-Verlag (2009)
10. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. FMSD 34, 238–304 (2009)
11. Lallali, M., Zaidi, F., Cavalli, A.: Timed modeling of web services composition for automatic testing. In: 2007 Third International IEEE Conference on Signal-Image Technologies and Internet-Based System. pp. 417–426. IEEE (2007)
12. Larsen, K.G., Mikucionis, M., Nielsen, B., Skou, A.: Testing real-time embedded software using Uppaal-Tron: An industrial case study. In: Proceedings of EMSOFT'05. pp. 299–306. ACM (2005)
13. Marre, B., Arnould, A.: Test sequences generation from LUSTRE descriptions: GATEL. In: Proceedings ASE'2000. pp. 229–237 (2000)
14. Merayo, M.G., Núñez, M., Rodríguez, I.: Formal testing from timed finite state machines. Computer networks 52(2), 432–460 (2008)
15. Mikucionis, M., Larsen, K.G., Nielsen, B.: T-Uppaal: Online model-based testing of real-time systems. In: Proceedings of ASE'2004. pp. 396–397. IEEE (2004)
16. Naito, S., Tsunoyama, M.: Fault detection for sequential machines by transition-tours. Proceedings of Fault Tolerant Computer Systems pp. 238–243 (1981)
17. Nguena Timo, O., Petrenko, A., Ramesh, S.: Multiple Mutation Testing from Finite State Machines with Symbolic Inputs. In: Proceedings of ICTSS'2017. pp. 354–375. Springer (2017)
18. Nguena Timo, O., Petrenko, A., Ramesh, S.: Checking Sequence Generation for Symbolic Input/Output FSMs by Constraint Solving. In: Proceedings of ICTAC'18. pp. 36–51. Springer (2018)
19. Nguena Timo, O., Rollet, A.: Conformance testing of variable driven automata. In: WFCS'2010. pp. 241–248. IEEE (2010)

20. Nilsson, R., Offutt, J., Mellin, J.: Test case generation for mutation-based testing of timeliness. Electr. Notes Theor. Comput. Sci. 164(4), 97–114 (2006)
21. Petrenko, A., Nguena Timo, O., Ramesh, S.: Multiple Mutation Testing from FSM. In: Proceedings of FORTE'2016. pp. 222–238. Springer (2016)
22. Petrenko, A., Yevtushenko, N.: Test Suite Generation from a FSM with a Given Type of Implementation Errors. In: Proceedings of the IFIP TC6/WG6.1 Twelfth International Symposium on Protocol Specification, Testing and Verification. pp. 229–243 (1992)
23. Raymond, P., Nicollin, X., Halbwachs, N., Waber, D.: Automatic testing of reactive systems. In: Proceedings of RTSS'98. pp. 200–209. IEEE (1998)
24. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT Solvers to Cryptographic Problems. In: Kullmann, O. (ed.) SAT 2009. pp. 244–257 (2009)
25. Tretmans, J.: Test generation with inputs, outputs, and repetitive quiescence. Software-Concepts and Tools 17, 103–120 (1996)
26. Tvardovskii, A., El-Fakih, K., Yevtushenko, N.: Deriving tests with guaranteed fault coverage for finite state machines with timeouts. In: Proceedings of ICTSS'2018. vol. 11146, pp. 149–154. Springer (2018)
27. Vega, J.J.O., Perrouin, G., Amrani, M., Schobbens, P.: Model-based mutation operators for timed systems: A taxonomy and research agenda. In: Proceedings of QRS'2018. pp. 325–332. IEEE (2018)
28. Zhigulin, M., Yevtushenko, N., Maag, S., Cavalli, A.R.: FSM-Based Test Derivation Strategies for Systems with Time-Outs. In: Proceedings of QSIC'2011. pp. 141–149 (2011)